Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Parallel Computing 29 (2003) 167–186

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Message passing and shared address space parallelism on an SMP cluster

Hongzhang Shan [a], Jaswinder P. Singh [b], Leonid Oliker [a], Rupak Biswas [c,*]

[a] *NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*
[b] *Department of Computer Science, Princeton University, Princeton, NJ 08544, USA*
[c] *NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035, USA*

Received 1 February 2001; received in revised form 25 September 2002; accepted 30 September 2002

Abstract

Currently, message passing (MP) and shared address space (SAS) are the two leading parallel programming paradigms. MP has been standardized with MPI, and is the more common and mature approach; however, code development can be extremely difficult, especially for irregularly structured computations. SAS offers substantial ease of programming, but may suffer from performance limitations due to poor spatial locality and high protocol overhead. In this paper, we compare the performance of and the programming effort required for six applications under both programming models on a 32-processor PC-SMP cluster, a platform that is becoming increasingly attractive for high-end scientific computing. Our application suite consists of codes that typically do not exhibit scalable performance under shared-memory programming due to their high communication-to-computation ratios and/or complex communication patterns. Results indicate that SAS can achieve about half the parallel efficiency of MPI for most of our applications, while being competitive for the others. A hybrid MPI + SAS strategy shows only a small performance advantage over pure MPI in some cases. Finally, improved implementations of two MPI collective operations on PC-SMP clusters are presented.

Keywords: PC cluster; Message passing; Distributed shared memory; Benchmark applications; Parallel performance

* Corresponding author. Tel.: +1-650-604-4411; fax: +1-650-604-3957.
E-mail addresses: hshan@lbl.gov (H. Shan), jps@cs.princeton.edu (J.P. Singh), loliker@lbl.gov (L. Oliker), rbiswas@nas.nasa.gov (R. Biswas).

0167-8191/02/$ - see front matter
PII: S0167-8191(02)00222-3

1. Introduction

The emergence of scalable computer architectures using clusters of PCs (or PC-SMPs) with commodity networking has made them attractive platforms for high-end scientific computing. Currently, message passing (MP) and shared address space (SAS) are the two leading programming paradigms for these systems. MP has been standardized with MPI, and is the more common and mature parallel programming approach. It provides both functional and performance portability; however, code development can be extremely difficult, especially for irregularly structured computations [15,16]. A coherent SAS has been shown to be very effective at moderate scales for a wide range of applications when supported efficiently in hardware [9,10,22–24]. The automatic management of naming and coherent replication in the SAS model also substantially eases the programming task compared to explicit MP, particularly for complex irregular applications that are becoming increasingly routine as multiprocessing matures. This programming advantage can often be translated directly into performance gains [24,25]. Even as hardware-coherent machines replace traditional distributed-memory systems at the high end, clusters of commodity PCs and PC-SMPs are becoming popular for scalable computing. On these systems, the MP paradigm is dominant while the SAS model is unproven since it is implemented in software. Given the ease of SAS programming, it is therefore important to understand its performance tradeoffs with MP on commodity cluster platforms.

Approaches to support SAS in software across clusters differ not only in the specialization and efficiencies of networks but also in the granularities at which they provide coherence. Fine-grained software coherence uses either code instrumentation [19,20] for access control or commodity-oriented hardware support [18] with the protocol implemented in software. Page-grained software coherence takes advantage of the virtual memory management facilities to provide replication and coherence at page granularity [12]. To alleviate false sharing and fragmentation problems, a relaxed consistency model is used to buffer coherence actions. Lu et al. [13] compared the performance of PVM and the TreadMarks page-based software shared-memory library on an 8-processor network of ATM-connected workstations and on an 8-processor IBM SP2. They found that TreadMarks generally performs slightly worse. Karlsson and Brorsson [11] compared the characteristics of communication patterns in MP and page-based software shared-memory programs, using MPI and TreadMarks running on an SP2. They found that the fraction of small messages in the TreadMarks executions lead to poor performance. However, the platforms used by both these groups were of much lower performance, smaller scale, and not SMP based. In addition, the protocols used for these experiments were quite inefficient. Recently, both the communication network and the protocols for shared virtual memory (SVM) have made great progress. Some GB/s networks have been put into use. A new SVM protocol, called GeNIMA [2], for page-grained SAS on clusters uses general-purpose network interface support to significantly reduce protocol overheads. It has been shown to perform quite well for medium-size systems on a fairly wide range of applications, achieving at least half the parallel efficiency of a

high-end hardware-coherent system and often exhibiting comparable behavior [2,8]. Thus, a study comparing the performance of using GeNIMA against the dominant way of programming for clusters today, namely MPI, becomes necessary and important.

In this paper, we compare performance of the MP and SAS programming models using the best implementations available to us (MPI/Pro from MPI Software Technology, for MPI, and the GeNIMA SVM protocol for SAS) on a cluster of eight 4-way SMPs (a total of 32 processors) running Windows NT 4.0. Our application suite includes codes that scale well on tightly coupled machines, as well as those that present a challenge to scalable performance because of their high communication-to-computation ratios and/or complex communication patterns. Our results show that if very high performance is the goal, the difficulty of MP programming appears to be necessary for commodity SMP clusters of today. Instead, if ease of programming is important, then SAS provides it at roughly a factor-of-two deterioration in performance for many applications, and somewhat less for others. This is encouraging for SVM, given the diverse nature of our application suite and the relative maturity of the MPI library. Application-driven research into coherence protocols and extended hardware support should reduce SVM and SAS overheads on future systems.

We also investigated a hybrid strategy by implementing SAS codes within each SMP while using MP among the SMP nodes. This allows codes to potentially benefit from both loop-level and domain-level parallelism. Although this hybrid programming model is the best mapping to our underlying architecture, it has the disadvantages of adversely affecting portability and increasing code complexity. Furthermore, results show only a small performance gain over the pure MPI versions for a subset of our applications. Finally, we present improved implementations of two MPI collective operations (`MPI_Allreduce` and `MPI_Allgather`) on PC-SMP clusters. Results show that these new algorithms achieve significant improvements over the default MPI/Pro implementation.

The remainder of this paper is organized as follows. Section 2 describes our PC cluster platform, and the implementation of the two programming models. The benchmark applications are briefly described in Section 3, as are the modifications that were made to improve their cluster performance. Performance results are presented and critically analyzed in Section 4. Section 5 explores new algorithms to efficiently implement two collective functions of MPI. Section 6 summarizes our key conclusions.

2. Platform and programming models

The platform used for this study is a cluster of eight 4-way 200 MHz Pentium Pro SMPs located at Princeton University. Each of the 32 processors has separate 8 KB data and instruction L1 caches, and a unified 4-way set-associative 512 KB L2 cache. Each of the eight nodes runs Windows NT 4.0, has 512 MB of main memory, and is connected to other nodes either by Myrinet [3] or Giganet [6]. The SAS and MP programming models are built in software on top of these two networks respectively. All

our MPI and SAS codes are compiled using the `cl` compiler provided by Microsoft Visual Studio 6.0 with the standard compilation options.

2.1. SAS programming model

Much research has been done in the design and implementation of SAS for clustered architectures, both at page and at finer fixed granularities through code instrumentation. Among the most popular ways to support a coherent SAS in software on clusters is page-based SVM. SVM provides replication and coherence at the page granularity by taking advantage of virtual memory management facilities. To alleviate problems with false sharing and fragmentation, SVM uses a relaxed memory consistency model to buffer coherence actions such as invalidations or updates, and postpones them until a synchronization point. Multiple writer protocols are used to allow more than one processor to modify copies of a page locally and incoherently between synchronizations, thereby reducing the impact of write-write false sharing and making the page consistent only when needed by applying `diffs` and `write` notices. Many distinct protocols have been developed which use different timing strategies to propagate `write` notices and apply the invalidations to pages. Recently, a new protocol for SVM called GeNIMA has been developed that shows good performance on moderate-scale systems for a wide spectrum of applications, achieving at least half the parallel efficiency of a high-end hardware-coherent machine [2,8]. It uses general-purpose network interface support to significantly improve protocol overheads. Thus, we select GeNIMA as our protocol for the SAS programming model. It is built on top of VMMC, a high-performance, user-level virtual memory mapped communication library [5]. VMMC itself runs on the Myrinet network [3].

The SMP nodes in our cluster are connected to a Myrinet system area network via a PCI bus. A single 16-way Myrinet crossbar switch is used to minimize contention in the interconnect. Each network interface has a 33 MHz programmable processor and connects the node to the network with two unidirectional links of 160 MB/s peak bandwidth. The actual node-to-network bandwidth, however, is constrained by the 133 MB/s PCI bus. The parallelism constructs and calls needed by the SAS programs are identical to those used in our hardware-coherent platform (SGI Origin2000) implementation [22–24], making portability trivial between these systems.

2.2. MP programming model

The MP implementation used in this work is MPI/Pro from MPI Software Technology, and is developed directly on top of Giganet networks [6] by the VIA [27] interface. By selecting MPI/Pro instead of building our own MPI library from VMMC, we can compare the best known versions of both programming models. In fact, MPI/Pro uses the underlying shared memory to communication within a single PC node. Thus our final conclusions are not affected by a potentially poor implementation of the communication layer. Fortunately, as shown in Table 1, VIA and VMMC have similar communication times for a range of message sizes on our cluster platform.

Table 1
Communication times (in μs) of different message sizes (in bytes) for the VMMC and VIA interfaces

	Message size						
	4	16	64	256	1024	4096	16384
VMMC (SAS)	10.9	11.2	15.1	20.0	34.2	80.1	210
VIA (MPI)	10.3	10.6	12.4	14.3	23.8	65.5	231

Giganet performs somewhat better for short messages while Myrinet has a small advantage for larger messages. There should thus be little performance difference for similar MPI implementations across these two networks. Note that the Giganet network interfaces are also connected together by a single crossbar switch.

3. Benchmark applications

Our application suite consists of codes used in previous studies to examine the performance and implementation complexity of various programming models on hardware-supported cache-coherent platforms [21–24]. These codes include regular applications (FFT, OCEAN, and LU) as well as irregularly structured applications (RADIX, SAMPLE, and N-BODY). FFT performs the challenging one-dimensional fast Fourier transform using the six-step FFT method. LU performs the blocked LU factorization of a dense matrix. OCEAN simulates eddy currents in an ocean basin. RADIX sorts a series of integer keys in ascending order using the radix algorithm, while SAMPLE uses the sample sort algorithm. N-BODY simulates the interaction of a system of bodies in three dimensions over a number of time steps, using the Barnes–Hut algorithm [1].

All six codes have either high communication-to-computation ratios or complex communication patterns, making scalable performance on cluster platforms a difficult task. FFT uses a non-localized but regular all-to-all personalized communication pattern to perform a matrix transposition; i.e., every process communicates with all others, sending different data across the network. OCEAN exhibits primarily nearest-neighbor patterns, but in a multigrid formation rather than on a single grid. LU uses one-to-many non-personalized communication. RADIX uses all-to-all personalized communication, but in an irregular and scattered fashion. In contrast, the all-to-all personalized communication in SAMPLE is much more regular. Finally, N-BODY requires all-to-all all-gather communication and demonstrates unpredictable send/receive patterns.

All the SAS implementations except N-BODY come from the SPLASH-2 suite with some additional optimizations [8,23]. FFT, LU, and SAMPLE were ported to our PC-SMP without any modifications. For RADIX, we used the improved version described in [23] where keys destined for the same processor are buffered together instead of being exchanged in a scattered fashion. Some changes were also made to the SPLASH-2 version of OCEAN to improve its shared-memory performance [8] on clusters. For example, the matrix was partitioned by rows across processors instead

of by blocks, and significant alterations were made to the data structures. The N-BODY code required major modifications since the original version suffered from the high overhead of synchronizations during the shared-tree building phase. A new tree building method, called Barnes-spatial [21], has been developed to completely eliminate the expensive synchronization operations.

All the MPI implementations were obtained by transforming the corresponding SAS codes using similar partitioning algorithms. Most of the MPI programs were available from our earlier work on the Origin2000 [22–24], and ported directly onto the PC cluster without any changes; however, OCEAN and RADIX required some modifications for better performance. In OCEAN, the matrix is now partitioned by rows instead of by blocks. This allows each processor to communicate only with its two neighbors, thus reducing the number of messages while improving the spatial locality of the communicated data. For example, by using row partitioning, the OCEAN speedup on 32 processors improved from 14.15 to 15.20 for a data set of 514×514 grid points.

For RADIX, in the key exchange stage, each processor now sends only one message to every other processor, containing *all* its chunks of keys destined for the destination processor. The receiving processor then reorganizes the data chunks to their correct positions. On a hardware-supported cache-coherent platform, a processor would send *each* contiguously-destined chunk of keys as a separate message, so that the data could be immediately inserted into the correct position by the receiver. However, this requires multiple messages from one processor to every other processor. Table 2 presents the RADIX speedups on the PC cluster and Origin2000 platforms using both messaging strategies for a data set of 32M integers. Notice that while the original multi-message implementation succeeds on the Origin2000 system (better speedups), the modified single-message approach is better suited for cluster platforms since reducing the number of messages at the cost of increased local computations is more beneficial. To study the two-level architectural effect (intra-node and inter-node), we also tested our applications by reorganizing the communication sequence in various ways (intra-node first, inter-node first, or intra-node and inter-node mixed). Interestingly, our results showed that the performance of the MPI programs was insensitive to the communication sequence.

All these applications have been previously used to evaluate the performance of MPI and SAS on the Origin2000 hardware-supported cache-coherent platform [22–24]. It was shown that SAS provides substantial ease of programming compared

Table 2
RADIX speedups on the PC cluster and Origin2000 using two messaging strategies for 32M integers

Origin2000 system				PC cluster			
Multi-message		Single-message		Multi-message		Single-message	
$P = 16$	$P = 32$	$P = 16$	$P = 32$	$P = 16$	$P = 32$	$P = 16$	$P = 32$
13.36	33.64	11.44	21.69	4.06	6.44	4.16	7.78

Table 3
Number of essential code lines for MPI and SAS implementations of our benchmark applications

	Benchmark application					
	FFT	OCEAN	LU	RADIX	SAMPLE	N-BODY
MPI	222	4320	470	384	479	1371
SAS	210	2878	309	201	450	950

to MP, while performance, though application-dependent, was sometimes better for SAS. The ease of programming holds true also on cluster systems, although some SAS code restructuring was required to improve performance. Nonetheless, an SAS implementation is still easier than MPI as has been argued earlier in the hardware-coherent context [10].

A comparison between MPI and SAS programmability is presented in Table 3. Observe that SAS programs require fewer lines of essential code (excluding the initialization and debugging code, and comments) compared to MPI. In fact, as application complexity (e.g., irregularity and dynamic nature) increases, we see a bigger reduction in programming effort using SAS. Note that "lines of code" is not considered a precise metric, but is nontheless a very useful measure of overall programming complexity. Some differences could also arise due to the programmer's style or experience with the programming models.

4. Performance analysis

In this section, we compare the performance of our benchmark applications under both the MP and SAS programming paradigms. For each application, parallel speedups and detailed time breakdowns are presented. To derive the speedup numbers, we use our best sequential runtimes for comparison. The parallel runtimes are decomposed into three components: LOCAL, RMEM, and SYNC. LOCAL includes CPU computation time and CPU waiting time for local cache misses, RMEM is the CPU time spent for remote communication, while SYNC represents the synchronization overhead. Two data set sizes are chosen for each application. The first is a *baseline* data set at which the SVM begins to perform "reasonably" well [8]. The second is a larger data set, since increasing the problem size generally tends to improve many inherent program characteristics, such as load balance, communication-to-computation ratio, and spatial locality.

4.1. FFT

The FFT algorithm has very high communication-to-computation ratio, which diminishes only logarithmically with problem size. It requires a non-localized but regular all-to-all personalized communication pattern to perform the matrix transposition, and cannot overlap the transposition and computation stages. In general, it is much more difficult to achieve high performance on the one-dimensional

Table 4
Speedups for the FFT application

	1M data set		4M data set	
	$P = 16$	$P = 32$	$P = 16$	$P = 32$
SAS	3.39	3.90	3.83	5.42
MPI	5.94	9.18	5.35	10.43

FFT, studied here, compared with higher-dimensional FFTs. Speedups for the SAS and MPI versions are presented in Table 4 for 1M and 4M data sets.

Neither MPI nor SAS show high scalability for our test cases. Increasing the data set size improves performance, but only slightly. This is mainly due to the pure communication of the transpose stage whose communication-to-computation ratio is not affected by problem size. In the sequential case, the transposition is responsible for approximately 16% of the overall runtime; however, it increases to 50% when using all 32 processors. It is inherently difficult to scale pure all-to-all communication. As the number of active processors increases, so does the contention in the network interface. Additionally, since each remote request requires access to the memory bus, increasing the number of processors has a deleterious effect on the local memory access time. This is particularly true for our commodity 4-way PC-SMP platform which suffers from high memory bus contention when all four processors simultaneously attempt to access memory. For example, the FFT LOCAL time (which includes the memory stall time) on two processors for the 4M data set is about 6 s. However, LOCAL drops to only about 4.8 s when all four processors are used, compared to an ideal of 3 s.

Observe though that the MPI implementation significantly outperforms SAS. To better understand the performance difference, Fig. 1 presents the time breakdown for the 4M data set running on 32 processors. We find that all the three time components (LOCAL, RMEM, and SYNC) are much larger in SAS than in MPI. In order to maintain page coherence, a high protocol overhead is introduced in SAS programs, including the time to compute `diffs`, creating timestamps, generating `write`

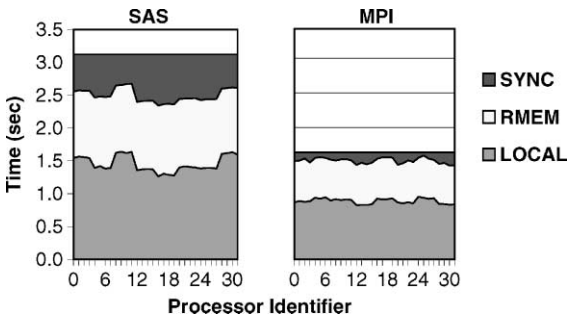

Fig. 1. FFT time breakdown for SAS and MPI on 32 processors for 4M data set.

notices, and performing garbage collection. This protocol overhead dramatically increases execution time while degrading local cache performance, thus causing a higher LOCAL time. In addition, the `diffs` generated for maintaining coherence immediately cause pages to be propagated to their home processors, thereby increasing network traffic and possibly causing more memory contention. Finally, at synchronization points, handling the protocol requirements causes a significant dilation of the synchronization interval, including the expensive invalidation of necessary pages. None of these protocol overheads exist in the MPI implementation. MPI does have the additional cost of packing and unpacking data for efficient communication; however, this overhead is incurred locally on the processors and is insignificant compared to the protocol costs associated with SAS.

One possible strategy to improve SAS performance would be to restructure the code so that the data structures more closely resemble the MPI implementation. For example, instead of allocating the matrix as a shared data structure, each submatrix that is transposed onto a different processor could be allocated separately. Unfortunately, this would dramatically increase the complexity of the SAS implementation, and thus sacrifice the programming ease of the shared-memory paradigm.

4.2. OCEAN

OCEAN exhibits a commonly used nearest-neighbor pattern, but in a multigrid rather than a single-grid formation. Parallel speedups are presented in Table 5. The scalability of the commodity SMP platform is relatively low, compared with previously obtained results on the hardware-supported cache-coherent architecture of the Origin2000 [22]. Although, the communication-to-computation ratio of OCEAN is high for small data sets, it quickly improves with larger problem sizes. This is especially true for the MPI version as shown in Table 5. Notice that SAS achieves superlinear speedup between 16 and 32 processors on the smaller data set. This occurs partly because as the number of processors increases, a larger fraction of the problem fits in cache.

The SAS implementation suffers from expensive synchronization overheads, as shown in Fig. 2. After each nearest-neighbor communication, a barrier synchronization is required to maintain coherence. Further analysis of the synchronization costs show that about 50% of this overhead is spent waiting, while the remainder is for protocol processing [2]. Thus, the synchronization cost can be improved either by

Table 5
Speedups for the OCEAN application

	258×258 grid		514×514 grid	
	$P = 16$	$P = 32$	$P = 16$	$P = 32$
SAS	2.17	5.96	5.44	6.49
MPI	4.97	8.03	7.45	15.20

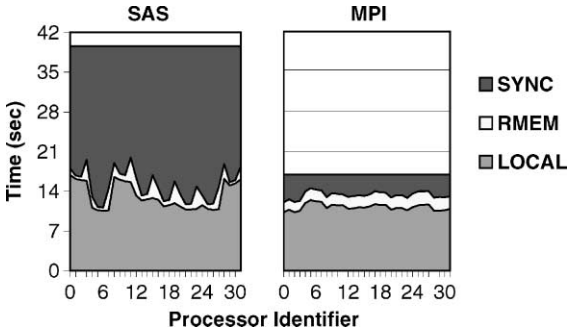

Fig. 2. OCEAN time breakdown for SAS and MPI on 32 processors for 514×514 grid size.

reducing protocol overhead or by increasing the data set size. Unfortunately, there is not enough computational work between synchronization points for the 514×514 problem size, especially because this grid is further coarsened into smaller subgrids during program execution. Moreover, OCEAN has a large memory requirement due to its use of more than 20 big data arrays, required for the multigrid code. Thus, we are prevented from running even larger data sets due to memory constraints. The synchronization within the MPI program is dramatically lower since it is implicitly implemented using send/receive pairs.

4.3. LU

The communication requirements of LU are smaller compared to our other benchmark codes, and thus we expect better performance for this application. This is confirmed by the results shown in Table 6. LU uses one-to-many non-personalized communication where the pivot block and the pivot row blocks are each communicated to $\sqrt{P}$ processors. From the time breakdown in Fig. 3, it is obvious that most of the overhead is in the LOCAL time. The LU performance could be further improved by reducing the synchronization cost caused by the load imbalance associated with the CPU wait time.

Notice that for LU, the performance of the SAS and MPI implementations are very close in both speedup and time breakdown characteristics. The protocol overhead of running the SAS version constitutes only a small fraction of the overall runtime. Unlike our FFT example, the LU matrix is organized in a four-dimensional

Table 6
Speedups for the LU application

	4096×4096 matrix		6144×6144 matrix	
	$P = 16$	$P = 32$	$P = 16$	$P = 32$
SAS	12.48	22.98	11.79	21.78
MPI	13.15	23.04	12.31	22.43

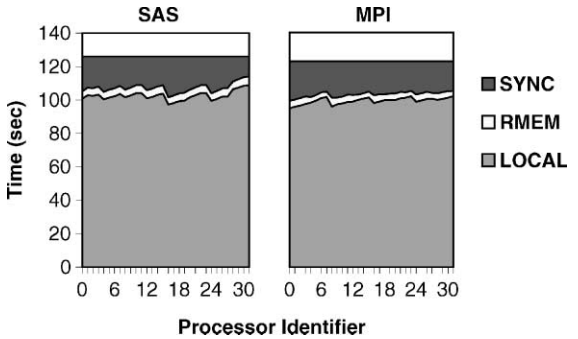

Fig. 3. LU time breakdown for SAS and MPI on 32 processors for 6144 × 6144 matrix size.

array such that blocks assigned to each processor are allocated locally and contiguously. Thus, each processor modifies only its own blocks, and the modifications are immediately applied to local data pages. As a result, no `diffs` generation and propagation are required, greatly reducing the protocol overhead. These performance results show that for applications with relatively low communication requirements, it is possible to achieve high scalability on commodity clusters using both MPI and SAS programming approaches.

4.4. RADIX

Unlike the three regularly structured codes (FFT, OCEAN, and LU) we have discussed so far, we now investigate three applications with irregular characteristics: RADIX, SAMPLE, and N-BODY. The RADIX sort benchmark requires all-to-all personalized communication, but in an irregular and scattered fashion. It also has a high communication-to-computation ratio that is independent of problem size and the number of processors. This application has large memory bandwidth requirements which can exceed the capacity of current SMP platforms; thus, high contention is caused on the memory bus when all four processors of a node are in use. The "aggregate" LOCAL time across processors is much greater than in the uniprocessor case, which leads to the poor performance shown in Table 7. However, MPI significantly outperforms the SAS implementation, since the latter has much larger RMEM and SYNC times as shown for the 32M integers data set in Fig. 4. These costs

Table 7
Speedups for the RADIX application

	4M integers		32M integers	
	$P = 16$	$P = 32$	$P = 16$	$P = 32$
SAS	1.33	1.66	1.86	2.70
MPI	3.78	5.67	4.16	7.78

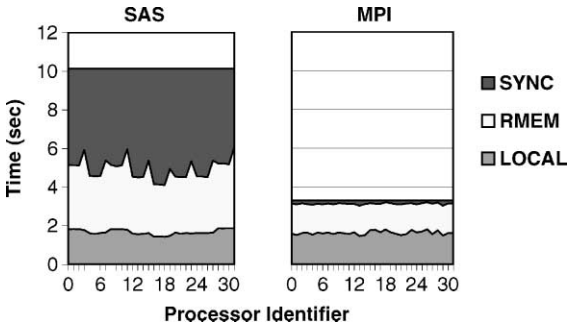

Fig. 4. RADIX time breakdown for SAS and MPI on 32 processors for 32M integers.

are due to the expensive protocol overhead of performing all-to-all communication, for reasons similar to those already discussed for FFT.

Note that choice of the proper implementation strategy for the MPI all-to-all communication is platform dependent. On the commodity cluster, each processor sends only one large message to all the other processors. The message contains all the data chunks required by the destination processor which, in turn, reorganizes the separate blocks of data into their correct positions. This is similar to the bucket sort algorithm used in the IS NAS Parallel Benchmark [14]. However, on the hardware-supported cache-coherent Origin2000, each processor sends the contiguous chunks of data directly to their destination processors in separate messages. Thus, unlike the cluster, each processor sends multiple messages to all the other processors in the system. The difference in these two approaches stems from the relatively high latency and low bandwidth of the cluster, where it is more efficient to send fewer messages in exchange for increased computational requirements of assembling the scattered data chunks.

4.5. SAMPLE

SAMPLE sorting also requires personalized all-to-all communication; however, it is less irregular than that for the RADIX algorithm. Speedups for SAMPLE are presented in Table 8, and compare favorably with the RADIX performance. Note that the same sequential time is used to compute the speedups for all the sorting codes.

Table 8
Speedups for the SAMPLE application

	4M integers		32M integers	
	$P = 16$	$P = 32$	$P = 16$	$P = 32$
SAS	2.10	2.13	4.97	4.89
MPI	4.89	8.60	5.73	11.07

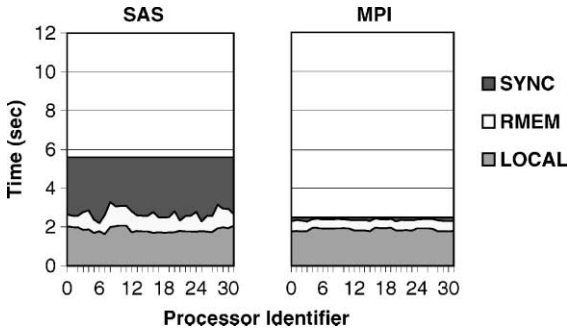

Fig. 5. SAMPLE time breakdown for SAS and MPI on 32 processors for 32M integers.

In SAMPLE, each processor first performs a local radix sort on its partitioned data. Next, an all-to-all communication is used to exchange keys, and a second local sort is conducted on the newly-received data. However, in the sequential case, only a single local sort is required. It is therefore reasonable to expect ideal SAMPLE performance to achieve only a 50% parallel efficiency.

Fig. 5 presents the time breakdown of SAMPLE for the larger data set on 32 processors. The *y*-axis scale is the same as in Fig. 4 for easier comparisons. Observe that the RMEM and SYNC times are significantly smaller than those of RADIX, for both MPI and SAS. As a result, the SAMPLE algorithm outperforms RADIX. Note that the LOCAL time for SAMPLE is only slightly greater than RADIX, even though much more computation is performed in SAMPLE. This indicates that contention on the memory bus for RADIX is higher than that for SAMPLE due to the greater irregularity of its memory access patterns. On the Origin2000, we found that RADIX performs better than SAMPLE in most cases; however, the reverse is true on our PC cluster. This result further verifies that reducing messages is much more important on a cluster platform than reducing the local computations.

4.6. N-BODY

Finally, we examine the performance of the N-BODY simulation. We use the Barnes–Hut [1] algorithm which employs a tree structure to reduce the complexity from $O(N)$ to $O(N \log N)$. Hence, tree building is an essential component of

Table 9
Speedups for the N-BODY application

	32K particles		128K particles	
	$P = 16$	$P = 32$	$P = 16$	$P = 32$
SAS	6.05	9.31	10.64	14.30
MPI	8.15	14.10	14.05	26.94

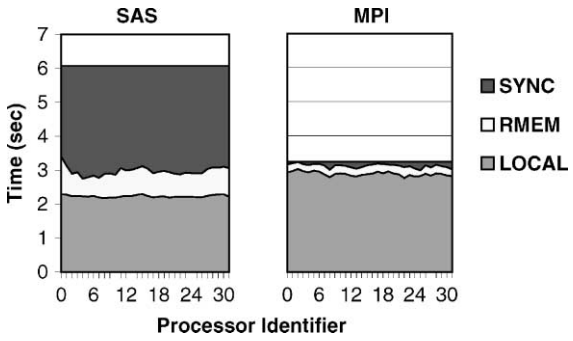

Fig. 6. N-BODY time breakdown for SAS and MPI on 32 processors for 128K particles.

the solution process. Table 9 shows that MPI once again outperforms SAS, especially for the larger data set. For 128 K particles on 32 processors, MPI achieves almost twice the performance of SAS.

The time breakdown for this larger data set on 32 processors is shown in Fig. 6. The SAS implementation has higher SYNC and RMEM times compared to MPI, but the synchronization overhead clearly dominates the overall runtime. This is because at each synchronization point, many `diffs` and `write` notices are processed by the coherence protocol. In addition, a large number of shared pages are invalidated. Further analysis shows that 82% of the barrier time is spent on protocol handling. This expensive synchronization overhead is incurred in all of our applications except LU, causing a degradation of SAS performance.

Unlike our other five applications, the MPI version of N-BODY has a higher LOCAL time than the SAS counterpart. This is due to the use of different high-level algorithms for each programming model. In the SAS implementation, each processor builds one part of a globally shared tree; while in MPI, a locally essential tree is created on each processor. Building the locally essential tree across distributed memories is much more complex than using a shared memory to build a single globally addressable tree. Therefore, there is a higher computational tree-building cost in the MPI implementation [24]. However, with large data sets, tree building becomes computationally insignificant compared to the other phases of the N-BODY simulation, most notably the force calculation.

4.7. Hybrid programming

All these results demonstrate that MP programming significantly outperforms SAS for our application suite. However, it is not obvious that MP within each SMP is the most effective use of the system. A recently proposed programming paradigm combines two layers of parallelism, by implementing SAS codes within each SMP while using MP among the SMP nodes. Here, the hardware directly supports cache coherence for the SAS code segments, while inter-SMP communication relies on the network through MP. This mixed-mode/hybrid programming strategy allows

Table 10
Runtimes (in s) for SAS, MPI, and hybrid MPI+SAS implementations of the benchmark applications on 32 processors for the larger data sets

	Benchmark application					
	FFT	OCEAN	LU	RADIX	SAMPLE	N-BODY
SAS	3.12	39.6	126	10.1	5.60	6.08
MPI	1.62	16.9	123	3.52	2.48	3.24
Hybrid	1.83	15.1	117	3.65	2.78	3.23

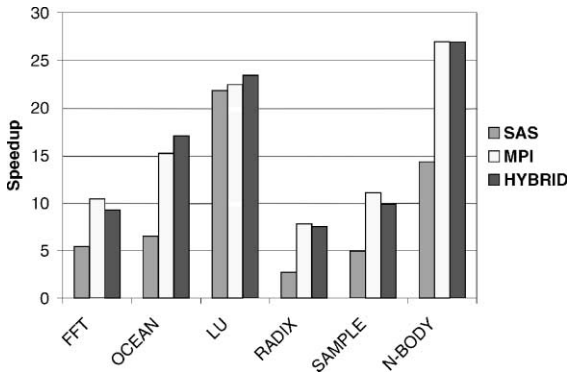

Fig. 7. Speedup comparison of the hybrid implementation with SAS and MPI on 32 processors for the larger data sets of the benchmark applications.

codes to potentially benefit from loop-level parallelism in addition to coarse-grained domain-level parallelism. Although this hybrid programming methodology is the best mapping to our underlying architecture, it remains unclear whether the performance gains of this approach compensate for its drawbacks.

Table 10 presents the hybrid MPI + SAS runtimes on 32 processors for the larger data sets of our application suite; Fig. 7 shows the corresponding speedup numbers. Overall, the hybrid implementation is within 12% of the pure MPI codes, and offers a small performance advantage in some cases. This is due to tradeoffs between the two approaches. For example, while SAS programming can potentially reduce the intra-SMP communication compared to MPI, it may require the additional overhead of explicit synchronizations. In addition, mixed-mode programming has the inherent disadvantages of adversely affecting portability and increasing code complexity. The latter is especially true for irregularly structured applications, such as the N-BODY simulation. Here, the hybrid implementation requires two types of tree-building algorithms: the MP version uses a distributed locally essential tree, while the SAS layer implements the additional data structure of a globally shared tree. Thus, in general, a pure MPI implementation is a more effective strategy than hybrid programming on SMP clusters. Similar conclusions have recently been drawn for other architectures and application domains [4,7,17].

5. MPI collective functions

An interesting question for clusters, particularly hybrid clusters of SMPs, is how to structure collective communication. In the MPI library, the communication operations can be divided into three broad categories: the basic send/receive functions, collective functions, and other operations. The performance of the basic send/receive operations primarily depends on the underlying communication hardware and the low-level software. On the other hand, the performance of the collective functions is affected by their individual implementations. Research in this area has been performed for a variety of platforms [26]. In this section, we discuss the algorithms suitable for our platform: a cluster of 4-way SMPs. Specifically, we explore the algorithms for two collective functions, `MPI_Allreduce` and `MPI_Allgather`, that are used in our applications. Here, we label the MPI/Pro implementation as "Original" (the exact algorithms used are not well documented) and use it as our baseline.

The most commonly used algorithm to implement `MPI_Allreduce` is the binary tree (B-Tree), shown in Fig. 8. The structure of our 4-way SMP nodes motivates us to modify the deepest level of the B-Tree to a quadtree structure, called B-Tree-4. Note that within an SMP node, the communication can be implemented either in shared memory or by using basic MPI send/receive functions; however, no measurable performance difference was observed between these two intra-node approaches. Timing results for reducing a double-precision floating-point variable per processor are shown in Table 11. The B-Tree implementation is about 7% faster than the MPI/Pro version, while the B-Tree-4 algorithm improves efficiency by another 5%. This

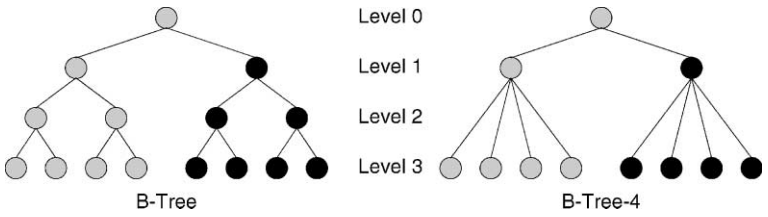

Fig. 8. The algorithms used to implement `MPI_Allreduce` on two 4-way SMP nodes.

Table 11
Execution times (in μs) for different `MPI_Allreduce` implementations on 32 processors (8 nodes) for one double-precision variable per processor

	Implementation algorithm		
	Original	B-Tree	B-Tree-4
One float	1117	1035	981

strategy can be easily extended to larger SMP nodes, leading to greater improvements in those cases.

We explored several different algorithms for the `MPI_Allgather` function. The first two methods were B-Tree and B-Tree-4 described above. In the B-Tree-4 algorithm, after the root processor (level 0 in Fig. 8) collects all the data, it broadcasts the data back to the processors at level 1 and below. Notice though that before the broadcast begins, each processor at level 1 already has a copy of all the data it collected from its own subtree. These two processors at level 1 can therefore directly exchange their data between themselves instead of sending them to the root processor and receiving them back. In other words, it is redundant to broadcast *all* the data to these processors at level 1. In fact, this idea can be extended to the lowest level of the tree (bounded by the size of an SMP node). We call this algorithm B-Tree-4$*$.

In Table 12, we present execution times of the different implementations of `MPI_Allgather` for both one and 1000 integers per processor. Results show that the B-Tree-4$*$ algorithm improves performance on our PC-SMP platform significantly, almost 27% and 41% compared to the original MPI/Pro implementation, for the two cases, respectively.

We applied these improved `MPI_Allreduce` and `MPI_Allgather` algorithms to four of the codes in our application suite. Table 13 presents the speedups with both the original MPI/Pro and our new implementations on 32 processors for the larger data sets of OCEAN, RADIX, SAMPLE, and N-BODY. Since most of the remote communication time in the applications is spent on the send/receive functions, the overall performance improves only slightly. SAMPLE shows the best results, achieving a 9% gain in performance. Codes relying heavily on these collective communication operations would obviously show a larger benefit. However, these results

Table 12
Execution times (in μs) for different `MPI_Allgather` implementations on 32 processors (8 nodes) for one and 1000 integers per processor

	Implementation algorithm			
	Original	B-Tree	B-Tree-4	B-Tree-4$*$
One integer	1538	1540	1479	1124
1K integers	1633	1052	993	975

Table 13
Speedups of four applications with the original and improved implementations of MPI collective functions on 32 processors for the larger data sets

	Benchmark application			
	OCEAN	RADIX	SAMPLE	N-BODY
Original	15.20	7.78	11.07	26.94
New	15.60	8.02	12.10	27.02

validate our improved implementations, and present future library developers with an approach for reducing collective communication overheads on SMP clusters.

6. Conclusions

In this paper, we studied the performance of and the programming effort required for six applications using the MP and SAS programming paradigms on a 32-processor PC-SMP cluster. The system consisted of eight 4-way Pentium Pro SMPs running Windows NT 4.0. To make a fair comparison between the two programming methodologies, we used the best known implementations of the underlying communication libraries. The MP version used MPI/Pro which is developed directly on top of Giganet by the VIA interface. The SAS implementation used the GeNIMA SVM protocol over the VMMC communication library, which runs on Myrinet. Experiments showed that VIA and VMMC have similar communication characteristics for a range of message sizes on our cluster platform.

Our application suite consisted of codes that typically do not exhibit scalable performance under shared-memory programming due to their high communication-to-computation ratios and/or complex communication patterns. Three regular applications (FFT, OCEAN, and LU) and three irregularly structured applications (RADIX, SAMPLE, and N-BODY) were tested. Porting these codes from the SGI Origin2000 system required some modifications to improve their performance on the cluster platform. Changes included reducing the number of messages in the MP versions, and removing fine-grained synchronizations from the SAS codes.

SAS provided substantial ease of programming, especially for the more complex applications which are irregular and dynamic in nature. However, unlike in a previous study on hardware-coherent machines where the SAS implementations were also performance-competitive with MPI, and despite all the research in SVM protocols and communication libraries in the last several years, SAS achieved only about half the parallel efficiency of MPI for most of our applications. The LU benchmark was an exception, in which the SAS implementation on the PC cluster showed very similar performance compared to the MPI version. The higher runtimes of the SAS codes were due to the excessive cost of the SVM protocol overhead associated with maintaining page coherence and implementing synchronizations. These costs include the time to compute `diffs`, create timestamps, generate `write` notices, and perform garbage collection. Future research should focus on reducing this synchronization cost. Possible approaches may include applying the `diffs` before the synchronization points, moving the shared-page invalidation operation out of synchronization points, and increasing the protocol hardware support.

We also investigated a hybrid MPI + SAS strategy that combined loop-level and domain-level parallelism. Even though this model naturally matched the architecture of our cluster platform, the results were only marginally better. Overall, our results demonstrated that if very high performance is the goal, the difficulty of MP programming appears to be necessary for commodity SMP clusters of today. On the other hand, if ease of programming is paramount, then SAS provides it at approxi-

mately a factor-of-two deterioration in performance for many applications, and somewhat less for others. This is encouraging for SVM, given the relative maturity of the MPI library and the diverse nature of our test suite. Finally, we presented new algorithms for improved implementations of MPI collective functions on PC clusters. Results showed significant gains compared to the default implementation.

Acknowledgements

The work of the first two authors was supported by NSF under grant number ESS-9806751 to Princeton University. The second author was also supported by PECASE and a Sloan Research Fellowship. The work of the third author was supported by the US Department of Energy under contract number DE-AC03-76SF00098.

References

[1] J.E. Barnes, P. Hut, A hierarchical O($N \log N$) force-calculation algorithm, Nature 324 (1986) 446–449.
[2] A. Bilas, C. Liao, J.P. Singh, Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems, in: Proceedings of the 26th International Symposium on Computer Architecture, Atlanta, GA, 1999, pp. 282–293.
[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.-K. Su, Myrinet: a gigabit-per-second local area network, IEEE Micro 15 (1995) 29–36.
[4] F. Cappello, D. Etiemble, MPI versus MPI + OpenMP on the IBM SP for the NAS benchmarks, in: Proceedings of the SC2000 Conference, Dallas, TX, 2000.
[5] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, K. Li, VMMC-2: efficient support for reliable, connection-oriented communication, in: Proceedings of the 5th Hot Interconnects Symposium, Stanford, CA, 1997.
[6] Giganet, Inc., Available from URL: http://www.giganet.com/.
[7] D.S. Henty, Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling, in: Proceedings of the SC2000 Conference, Dallas, TX, 2000.
[8] D. Jiang, B. O'Kelly, X. Yu, S. Kumar, A. Bilas, J.P. Singh, Application scaling under shared virtual memory on a cluster of SMPs, in: Proceedings of the 13th International Conference on Supercomputing, Rhodes, Greece, 1999, pp. 165–174.
[9] D. Jiang, H. Shan, J.P. Singh, Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors, in: Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming, Las Vegas, NV, 1997, pp. 217–229.
[10] D. Jiang, J.P. Singh, Scaling application performance on cache-coherent multiprocessors, in: Proceedings of the 26th International Symposium on Computer Architecture, Atlanta, GA, 1999, pp. 305–316.
[11] S. Karlsson, M. Brorsson, A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2, in: Proceedings of the 2nd International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing, Las Vegas, NV, 1998, pp. 189–201.
[12] K. Li, P. Hudak, Memory coherence in shared virtual memory systems, ACM Transactions on Computer Systems 7 (1989) 321–359.
[13] H. Lu, S. Dwarkadas, A.L. Cox, W. Zwaenepoel, Quantifying the performance differences between PVM and TreadMarks, Journal of Parallel and Distributed Computing 43 (1997) 65–78.

[14] NAS parallel benchmarks, Available from URL: http://www.nas.nasa.gov/Software/NPB/.
[15] L. Oliker, R. Biswas, Parallelization of a dynamic unstructured algorithm using three leading programming paradigms, IEEE Transactions on Parallel and Distributed Systems 11 (2000) 931–940.
[16] L. Oliker, R. Biswas, H.N. Gabow, Parallel tetrahedral mesh adaptation with dynamic load balancing, Parallel Computing 26 (2000) 1583–1608.
[17] L. Oliker, X. Li, P. Husbands, R. Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, SIAM Review 44 (2002) 373–393.
[18] S.K. Reinhardt, J.R. Larus, D.A. Wood, Tempest and typhoon: user-level shared memory, in: Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL, 1994, pp. 325–336.
[19] D.J. Scales, K. Gharachorloo, C.A. Thekkath, Shasta: a low overhead, software-only approach for supporting fine-grain shared memory, in: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996, pp. 174–185.
[20] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, D.A. Wood, Fine-grain access control for distributed shared memory, in: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1994, pp. 297–306.
[21] H. Shan, J.P. Singh, Parallel tree building on a range of shared address space multiprocessors: algorithms and application performance, in: Proceedings of the 12th International Parallel Processing Symposium, Orlando, FL, 1998, pp. 475–484.
[22] H. Shan, J.P. Singh, A comparison of MPI, SHMEM and cache-coherent shared address space programming models on a tightly-coupled multiprocessor, International Journal of Parallel Programming 29 (2001) 283–318.
[23] H. Shan, J.P. Singh, Parallel sorting on cache-coherent DSM multiprocessors, in: Proceedings of the SC99 Conference, Portland, OR, 1999.
[24] H. Shan, J.P. Singh, L. Oliker, R. Biswas, A comparison of three programming models for adaptive applications on the Origin2000, Journal of Parallel and Distributed Computing 62 (2002) 241–266.
[25] J.P. Singh, A. Gupta, M. Levoy, Parallel visualization algorithms: Performance and architectural implications, IEEE Computer 27 (1994) 45–55.
[26] S. Sistare, R. vandeVaart, E. Loh, Optimization of MPI collectives on clusters of large-scale SMPs, in: Proceedings of the SC99 Conference, Portland, OR, 1999.
[27] Virtual interface architecture, Available from URL: http://www.viarch.org/.